CS-320

Computer Language Processing

Exercise Session 5

November 8, 2017

Overview

In today's exercise you will

- *perform type inference* on some examples seen in the lecture,
- see how we can actually *encode pairs* using functions, and
- learn about some simple forms of *subtyping*.

Recap: Type-inference

In this week's lecture we heard about type inference, which allows us to enjoy the safety of type systems without requiring explicit type annotations in the program.

We saw anonymous functions:

$(x \Rightarrow x)$

This one corresponds to the identity function:

def id(x) = x

Using type inference we can find a function's *most general type*:

$$\forall X.\ X \to X$$

Such *type schemata* can be instantiated with different types:

if (id(**true**)) id(1) **else** id(2)

Finding the most general type

Exercise 1

- Find the most general type for each of the following functions:

def S(x, y, z) = (x(z))(y(z))

def cm(f, g) = x ⇒ f(g(x))

def cr(f) = x ⇒ (y ⇒ f(x,y))

def uncr(f) =
  p ⇒ (f(P1(p)))(P2(p))

def pr(x, y) = c ⇒ (c(x))(y)

def c1(p) = p(x ⇒ (y ⇒ x))

def c2(p) = p(x ⇒ (y ⇒ y))

def e(x, y) = c1(pr(x,y))

Pairs

Recall that in the last lecture we had a language with pairs. That is, we had pair types

$$T_1 \times T_2$$

constructors for pair values

$$(t_1, t_2)$$

and extractors for the first and the second component

$$fst(p) \quad snd(p)$$

Pairs

Recall that in the last lecture we had a language with pairs. That is, we had pair types

$$T_1 \times T_2$$

constructors for pair values

$$(t_1, t_2)$$

and extractors for the first and the second component

$$\mathit{fst}(p) \quad \mathit{snd}(p)$$

It turns out that in an untyped language with anonymous functions one can already *encode* pairs.
$\Rightarrow$ What about our typed language with anonymous functions?

Encoding pairs

Let us attempt to encode pairs in our language.

We will represent a pair as a function of type

$$Pair[A, B] = (A \to B \to C) \to C$$

where the C will be the result type of the computation depending on the pair.

Encoding pairs

Let us attempt to encode pairs in our language.

We will represent a pair as a function of type

$$Pair[A, B] = (A \to B \to C) \to C$$

where the C will be the result type of the computation depending on the pair.

We define functions to create pairs and extract their components:

```
def mkPair(a, b) = (f ⇒ f(a, b))
def fst(p) = p(a ⇒ b ⇒ a)
def snd(p) = p(a ⇒ b ⇒ b)
```

Encoding pairs

Exercise 2

```
def mkPair(a, b) = (f ⇒ f(a, b))
def fst(p) = p(a ⇒ b ⇒ a)
def snd(p) = p(a ⇒ b ⇒ b)
def f(x, y) = true
```

Consider the following program along with the above definitions:

f(fst(mkPair(1, **true**)), snd(mkPair(1, **true**)))

▷ What is the result of type inference on this program?

Encoding pairs

Exercise 2

```
def mkPair(a, b) = (f ⇒ f(a, b))
def fst(p) = p(a ⇒ b ⇒ a)
def snd(p) = p(a ⇒ b ⇒ b)
def f(x, y) = true
```

Consider the following program along with the above definitions:

```
f(fst(mkPair(1, true)), snd(mkPair(1, true)))
```

▷ What is the result of type inference on this program?

Now consider we only create a single pair:

```
def g(p) = f(fst(p), snd(p))
g(mkPair(1, true))
```

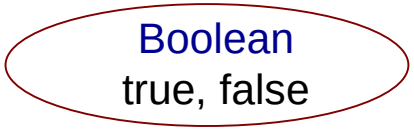
▷ What does type inference yield in this case?

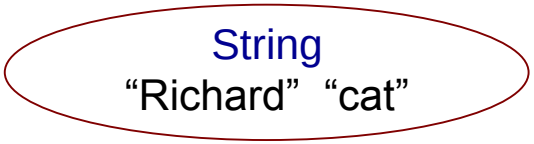
Meaning of Types

- Operational view: Types are named entities
  - such as the primitive types (Int, Bool etc.) and explicitly declared classes, traits …
  - their meaning is given by methods they have
  - constructs such as inheritance establish relationships between classes
- Mathematically, Types are sets of values
  - Int = { …, -2, -1, 0, 1, 2, … }
  - Boolean = { false, true }
  - Int → Int = { f : Int -> Int | f is computable }

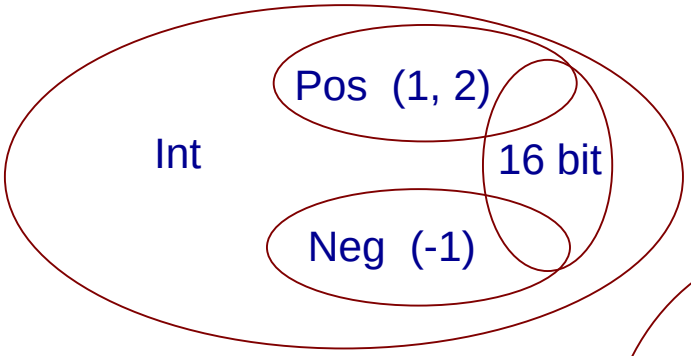
Types as Sets

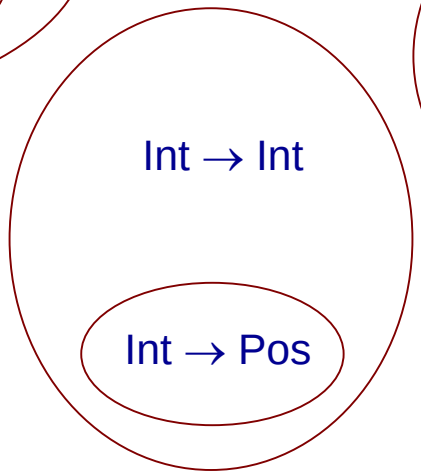
- Sets so far were disjoint



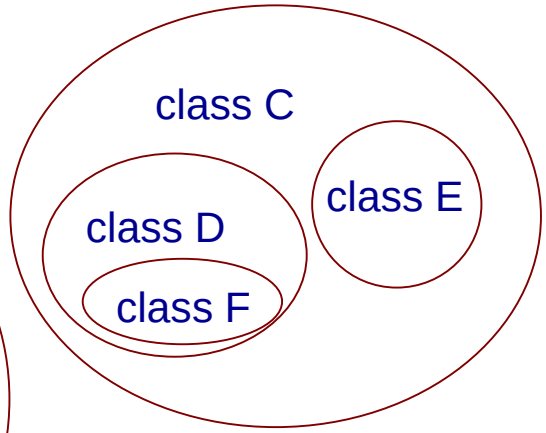


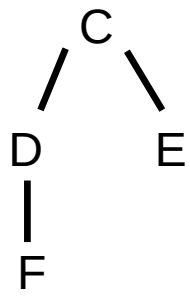
- Sets can overlap





C represents not only declared C, but all possible extensions as well





F extends D,
D extends C

SUBTYPING

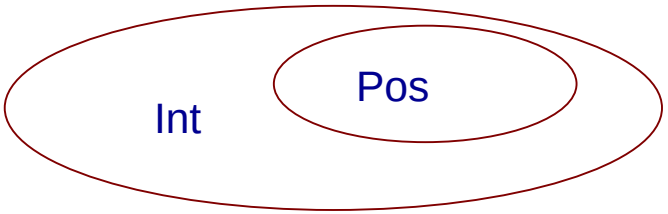
Subtyping

- Subtyping corresponds to subset
- Systems with subtyping have non-disjoint sets
- $T_1 <: T_2$ means T_1 is a subtype of T_2
  - corresponds to $T_1 \subseteq T_2$ in sets of values
- Rule for subtyping: analogous to set reasoning

$$\frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma \vdash e : T_2}$$

In terms of sets

$$\frac{e \in T_1 \qquad T_1 \subseteq T_2}{e \in T_2}$$



Types for Positive and Negative Ints

Int = { ... , -2, -1, 0, 1, 2, ... }

Pos = { 1, 2, ... } (not including zero)

Neg = { ..., -2, -1 } (not including zero)

types:

Pos <: Int

Neg <: Int

$$\frac{\Gamma \vdash x\colon \mathrm{Pos} \qquad \Gamma \vdash y\colon \mathrm{Pos}}{\Gamma \vdash x + y\colon \mathrm{Pos}}$$

$$\frac{\Gamma \vdash x\colon \mathrm{Pos} \qquad \Gamma \vdash y\colon \mathrm{Neg}}{\Gamma \vdash x * y\colon \mathrm{Neg}}$$

$$\frac{\Gamma \vdash x\colon \mathrm{Pos} \qquad \Gamma \vdash y\colon \mathrm{Pos}}{\Gamma \vdash x / y\colon \mathrm{Pos}}$$

sets:

Pos $\subseteq$ Int

Neg $\subseteq$ Int

$$\frac{x \in \mathrm{Pos} \qquad y \in \mathrm{Pos}}{x + y \in \mathrm{Pos}}$$

$$\frac{x \in \mathrm{Pos} \qquad y \in \mathrm{Neg}}{x * y \in \mathrm{Neg}}$$

$$\frac{x \in \mathrm{Pos} \qquad y \in \mathrm{Pos}}{x / y \in \mathrm{Pos}}$$

(y not zero)

(x/y well defined)

More Rules

$$\frac{\Gamma \vdash \mathrm{x}\colon \mathrm{Neg} \qquad \Gamma \vdash \mathrm{y}\colon \mathrm{Neg}}{\Gamma \vdash \mathrm{x} * \mathrm{y}\colon \mathrm{Pos}}$$

$$\frac{\Gamma \vdash \mathrm{x}\colon \mathrm{Neg} \qquad \Gamma \vdash \mathrm{y}\colon \mathrm{Neg}}{\Gamma \vdash \mathrm{x} + \mathrm{y}\colon \mathrm{Neg}}$$

More rules for division?

$$\frac{\Gamma \vdash \mathrm{x}\colon \mathrm{Neg} \qquad \Gamma \vdash \mathrm{y}\colon \mathrm{Neg}}{\Gamma \vdash \mathrm{x} \,/\, \mathrm{y}\colon \mathrm{Pos}}$$

$$\frac{\Gamma \vdash \mathrm{x}\colon \mathrm{Pos} \qquad \Gamma \vdash \mathrm{y}\colon \mathrm{Neg}}{\Gamma \vdash \mathrm{x} \,/\, \mathrm{y}\colon \mathrm{Neg}}$$

$$\frac{\Gamma \vdash \mathrm{x}\colon \mathrm{Int} \qquad \Gamma \vdash \mathrm{y}\colon \mathrm{Neg}}{\Gamma \vdash \mathrm{x} \,/\, \mathrm{y}\colon \mathrm{Int}}$$

Making Rules Useful

- Let x be a variable

$$\frac{\Gamma \vdash \mathrm{x}\colon \mathrm{Int} \qquad \Gamma \oplus \{(x, Pos)\} \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash (\mathrm{if}\ (\mathrm{x} > 0)\ e_1\ \mathrm{else}\ e_2)\colon \mathrm{T}}$$

$$\frac{\Gamma \vdash \mathrm{x}\colon \mathrm{Int} \qquad \Gamma \vdash e_1 : T \qquad \Gamma \oplus \{(x, Neg)\} \vdash e_2 : T}{\Gamma \vdash (\mathrm{if}\ (\mathrm{x} >= 0)\ e_1\ \mathrm{else}\ e_2)\colon \mathrm{T}}$$

```
var x : Int
var y : Int
if (y > 0) {
  if (x > 0) {
    var z : Pos = x * y
    res = 10 / z
} }
```

type system proves: no division by zero

Subtyping Example

```
def f(x:Int) : Pos = {
  if (x < 0) -x else x+1
}
var p : Pos
var q : Int
q = f(p)
```

Does this statement type check?

Given:

$$\text{Pos} <: \text{Int}$$

$$\Gamma \vdash \text{f: Int} \rightarrow \text{Pos}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{p: Pos} \qquad \text{Pos} <: \text{Int}}{\text{p: Int}} \qquad \text{f: Int} \rightarrow \text{Pos}}{\text{f(p): Pos}} \qquad \text{Pos} <: \text{Int}}{\text{f(p): Int}} \qquad (\text{q, Int}) \in \Gamma}{\text{q=f(p): void}}$$

Subtyping Example

```
def f(x:Pos) : Pos = {
  if (x < 0) -x else x+1
}
var p : Int
var q : Int
q = f(p)
```

Does this statement type check?

does not type check

What Pos/Neg Types Can Do

```
def multiplyFractions(p1 : Int, q1 : Pos, p2 : Int, q2 : Pos) : (Int,Pos) {
  (p1*q1, q1*q2)
}
def addFractions(p1 : Int, q1 : Pos, p2 : Int, q2 : Pos) : (Int,Pos) {
  (p1*q2 + p2*q1, q1*q2)
}
def printApproxValue(p : Int, q : Pos) = {
  print(p/q)   // no division by zero
}
```

More sophisticated types can track intervals of numbers and ensure that a program does not crash with an array out of bounds error.

Intersection types

(Exam question 2015)

$\Rightarrow$ Dedicated exercise sheet.

Problem 4: Intersection Types (25 points)

In this exercise, we will consider the notion of intersection of types. Let T_1 and T_2 be two types belonging to our langauge. An expression has an intersection type $T_1 \wedge T_2$ iff it can be typed as both T_1 and T_2. Therefore, we have the following type rules.

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma \vdash e : T_2}{\Gamma \vdash e : T_1 \wedge T_2} \qquad \frac{\Gamma \vdash e : T_1 \wedge T_2}{\Gamma \vdash e : T_1} \qquad \frac{\Gamma \vdash e : T_1 \wedge T_2}{\Gamma \vdash e : T_2}$$

We consider $T_1 \wedge T_2$ and $T_2 \wedge T_1$ to be the same. In the above rules T_1 and T_2 can also be function types like $R \to S$.

a) **[5 pts]** If T_1 and T_2 are arbitrary types, consider the following three expressions denoting types: $T_1 \wedge T_2$, T_1, and T_2. State all subtyping relations that you believe should hold among the 3×3 possible pairs of expressions.

Enter `<:` if the type corresponding to the row is a subtype of the type corresponding to the column; enter / if this is not necessarily the case.

<:	T_1	T_2	$T_1 \wedge T_2$
T_1			
T_2			
$T_1 \wedge T_2$			

In the next part of the exercise, you are required to come up with a type derivation involving intersection types. Consider a language, similar to the one descried in lecturecise 12, with arithmetic operations, if-else expressions, assignment and block statements, that has the following types: Pos, Neg, Int and $Bool$. (We provide all the type rules that you may need for this exercise at the end of this question.)
Consider the function f shown below. Γ_0 is the initial type environment before the beginning of the function.

$\Gamma_0 = \{$ divk : $(Pos \to Pos) \wedge (Neg \to Neg)\}$

```
def f(x : Int) : Int {
  if(x > 0) divk(x)
  else
   if(x < 0) divk(x)
   else x
}
```

`divk` is a function (e.g. like $x => 10/x$) that maps positive integers to positive integers and negative integers to negative integers. Observe that with intersection types we can type the function as $(Pos \to Pos) \wedge (Neg \to Neg)$.

b) **[20 pts]** Complete the type derivation for the body of the function f, shown on page 5, by filling in the holes marked with ?. If the expression will not type check, show the step where the type derivation cannot proceed. You will need to use only the type rules of intersection types and the types rules shown below.

Type Rules:

$$\frac{(x,T) \in \Gamma}{\Gamma \vdash x : T} \qquad Pos <: Int \quad Neg <: Int \qquad \frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma \vdash e : T_2}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \{e\} : T} \qquad \frac{s_1 : Unit \qquad \Gamma \vdash \{s_2; \cdots s_n\} : T}{\Gamma \vdash \{s_1; \cdots ; s_n\} : T} \qquad \frac{\Gamma \oplus \{(x,T)\} \vdash \{s_2; \cdots s_n\} : T}{\Gamma \vdash \{\text{var } x : T; s_2; \cdots ; s_n\} : T}$$

$$\frac{\Gamma \vdash x : Int \qquad \Gamma \oplus \{(x, Pos)\} \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{if}(x > 0)\ e_1\ \text{ else }\ e_2 : T}$$

$$\frac{\Gamma \vdash x : Int \qquad \Gamma \oplus \{(x, Neg)\} \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{if}(x < 0)\ e_1\ \text{ else }\ e_2 : T} \qquad \frac{\Gamma \vdash b : Bool \qquad \Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{if}(b)\ e_1\ \text{ else }\ e_2 : T}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \cdots \qquad \Gamma \vdash e_n : T_n \qquad \Gamma \vdash g : (T_1 \times \cdots \times T_n) \to T}{\Gamma \vdash g(e_1, \cdots, e_n) : T}$$

$$
\frac{
\dfrac{
\dfrac{\Gamma_1 \vdash x : Pos \qquad \dfrac{?}{\Gamma_1 \vdash \texttt{divk} : ?}}{?}
}{\Gamma_1 \vdash \texttt{divk(x)} : Int}
\qquad
\dfrac{
\Gamma \vdash x : Int \qquad
\dfrac{
\dfrac{\Gamma_2 \vdash x : Neg \qquad \dfrac{?}{?}}{?}
}{\Gamma_2 \vdash \texttt{divk(x)} : ?}
}{\Gamma \vdash \texttt{if(x < 0) divk(x) else x} : Int}
}{\Gamma \vdash \texttt{if(x > 0) divk(x) else if(x < 0) divk(x) else x} : Int}
$$

where,

$\Gamma_1 = ?$

$\Gamma_2 = ?$